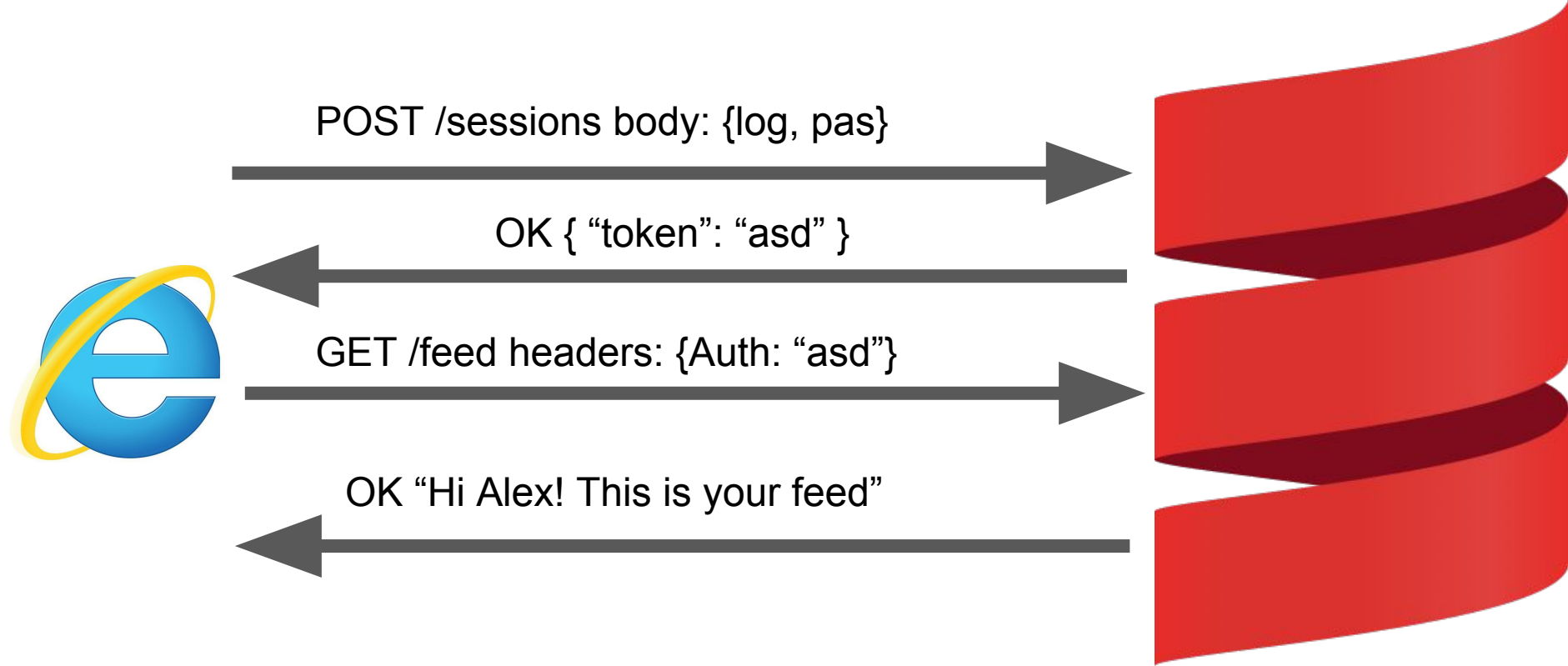


Twitter - continue

Brief overview

- Teams of 2 or 3 members
- API only
- No DB
- Authentication and Authorization
- Each user has own feed, consists of other users messages/retweets
- User can write tweets, subscribe to other users and retweet
- User can edit/delete only his own tweets

Tokens for authentication



JWT for tokens

JWT - JSON Web Tokens

It is simply encoded JSON using base64

Consists of “header.payload.signature”

Payload is info we want to store on client :

```
{  
  "userId": "1",  
  "admin": true  
}
```

More on JWT structure

```
// To build signature we need hash-fun HMAC_MD5(key, data)
HMAC_MD5(K, D) = MD5(K + MD5(K + D))
data = base64(header) + "." + base64(payload)
secret = "some secret string used for crypto-things"
signature = HMAC_MD5(data, secret) // produces hash
```

More on JWT structure

Header is meta-info about our token:

```
{  
  "alg": "HS256", // HS256 = HMAC + SHA256  
  "typ": "JWT"    // type of token  
}
```

Good article about algorithms, JWT and choosing right key:

<https://goo.gl/VebAmC>

JWT result

So JWT token = base64(header) + “.” + base64(payload) + “.” + signature

User and server can read header and payload, but when something is modified server can distinguish that.

Server can verify token because it knows the key used when building signature.

! As everyone can read header and payload do not store sensitive data there

Payload usage example



POST /sessions body: {log, pas}


OK { "token": "asd.payload.dfg" }

```
"payload" : {  
  "userId": "1",  
  "timestamp": "2018-02-05-17-00",  
  "expTime" : "24h"  
}
```



Payload usage example

GET /feed headers: {Auth: "asd.payload.dfg"}



```
token = decode(header("Authentication", req))  
\\ unauthorised if token not decoded  
verifyTime(token)  
\\ unauthorised if expired  
user = find_user(token.payload.user_id)  
respond(feed(user))
```

OK "Hi Alex! This is your feed"

For-comprehension

```
case class User(val name: String, val bot: Boolean = false)
```

```
val userBase = List(new User("Saha"),  
    new User("Notification bot", true),  
    new User("Jennifer"),  
    new User("Dennis"))
```

We want to get users that are not bots

Let's do it in a straightforward way

```
for (i <- 0 to userBase.length - 1) {  
  if (!(userBase(i).bot)) {  
    println(userBase(i))  
  }  
}
```

That would be hell if we want to get a list instead of just
print:(

We know that we can get users in for

```
for (u <- userBase) {  
  if (!(u.bot)) {  
    println(u)  
  }  
}
```

Now looks much better, but we can do more

High order functions and for-comprehension

```
for (u <- userBase if !u.bot) println(u)
```

```
// desugared for-comprehension
```

```
userBase.filter(u => !u.bot).foreach(println)
```

Two ways in two lines - cool!

Tournament table

```
var res = ListBuffer[(User, User)]()

for (u1 <- userBase) {
  for (u2 <- userBase) {
    if (u1 != u2) {
      val tup = (u1, u2)
      res += tup
    }
  }
}

println(res.toList)
```

For-comprehension version

```
val res1 = for (u1 <- userBase; u2 <- userBase if (u1 != u2)) yield (u1, u2)
```

```
val res1 = for {  
  u1 <- userBase  
  u2 <- userBase  
  if (u1 != u2)  
} yield (u1, u2)
```

```
println(res1)
```

Desugared magic

```
def flatMap[A] (List[A], A => List[A]): List[A]
```

```
val l = List(1, 2)
```

```
flatMap(l, el => List(el, 0)) == List(1, 0, 2, 0)
```

```
val res1 = userBase.flatMap {  
    u1 => userBase  
        .filter(u2 => (u1 != u2))  
        .map(u2 => (u1, u2))  
}
```


Which functions used in for-comprehension

```
def foreach[A] (List[A], A => Unit): Unit
```

```
def map[A, B] (List[A], A => B): List[B]
```

```
def flatMap[A, B] (List[A], A => List[B]): List[B]
```

```
def filter[A] (List[A], A => Boolean): List[A]
```

Any for-comprehension could be done using this functions

Some comparisons

```
for (v <- values) println(v)  
values.foreach(v => println(v))
```

More general

```
for(x <- c1; y <- c2; z <-c3) {...}  
c1.foreach(x => c2.foreach(y => c3.foreach(z => {...})))
```

Some comparisons

```
for (v <- values) yield (v, 0)
```

```
values.map(v => (v, 0))
```

More general

```
for(x <- c1; y <- c2; z <- c3) yield {...}
```

```
c1.flatMap(x => c2.flatMap(y => c3.map(z => {...})))
```

Some comparisons

```
for(v <- values; if v > 5) yield v
```

```
values.filter(v => v > 5).map(v => v)
```

```
for(x <- c; if cond) yield {...}
```

```
c.filter(x => cond).map(x => {...})
```

Some comparisons

```
for (v <- values; v1 = v * 100) yield (v, v1 + 10)
```

```
values.map(v => (v, v * 100)).map { case (v, v1) => (v, v1 + 10)  
}
```

```
for (x <- c; y = ...) yield {...}
```

```
c.map(x => (x, ...)).map((x, y) => {...})
```

For-comprehension could be used outside of lists

```
for {  
  user      <- findUserInDB(1)  
  messages <- makeRequetToSomeService(user)  
} yield messages
```

:O :O :O :O :O :O We will use this cool feature later!

Basically List could be just abstracted

```
def foreach[A] (List[A], A => Unit): Unit
```

```
def map[A, B] (List[A], A => B): List[B]
```

```
def flatMap[A, B] (List[A], A => List[B]): List[B]
```

```
def filter[A] (List[A], A => Boolean): List[A]
```

```
def foreach[L, A] (L[A], A => Unit): Unit
```

```
def map[L, A, B] (L[A], A => B): L[B]
```

```
def flatMap[L, A, B] (L[A], A => L[B]): L[B]
```

```
def filter[L, A] (L[A], A => Boolean): L[A]
```

Resources

<https://docs.scala-lang.org/tour/for-comprehensions.html>

<https://docs.scala-lang.org/tutorials/FAQ/yield.html>

<https://jwt.io/>

<https://goo.gl/VebAmC>